Python

Python interpreter runs each file from top-to-bottom - if it sees a statement, it runs it! There is no default main() function that runs! It is conventional though, to have a main() in a Python program

```
def lets_go(to):
    print('Hey, ' + to)
    print("Let's go to Del Taco!")

def finish_food():
    print('Oh crap, now I have gas!')

def main():
    lets_go('Carey')
    print('Nom Nom')
    finish_food()

if __name__ = "__main__":
    main()
```

`So __name__ "__main__"`

- So it sets a special variable **name** to "**main**" which tells us that we should run our main function

Python's runtime type-checker detects that the types of values that a nad b refer to are incompatible, and generates an error as the operation is performed

Dynamic typing - the language chedcks the types of operands the moment an operation occurs. Static typing - bugs are found at compile time

In python, declaring a variable in an if block still keeps it in scope outside of the if-block

Classes

```
class Car:
    MILES_PER_GAL = 30
    def __init__(self, gallons):
        self.gas_gallons = gallons
        self.odometer = 0
    def drive(self, miles):
        gals_needed = miles / Car.MILES_PER_GAL
        if self.gas_gallons > gals_needed:
            self.gas_gallons -= gals_needed
            self.odometer += miles

    def get_odometer(self):
        return self.odometer

def use_car():
    c1 = Car(16)
    c1.drive(10)
    print(f'I drive {c1.get_odometer()} mi')
```

To define a constructor: `__init__` The constructor is called when you define a new object To define member variables in a python class:

- There is no way to declare a class's member variables without assigning them method is different from a normal python function because it is defined with self. and thus it can access member variables/fields.

If we changed

```
def get_odometer(self):
    return odometer
```

- If we remove the "self" prefix, this no longer accesses the member variable and so now it refers to a (nonexistent) local variable. So if we want to access the member variable in the constructor, we have to use the self keyword, it also technically makes it global to the object

Adding two __ before the method makes the function private

The definition of a python class is just a regular sequence of statements like a regular Python program! In fact, you can include any legal python code in your class definition.

Allocation of Objects: Python vs. C++ In C++, we can define an object on the stack or on the heap

```
Circle c(10);
Circle *p = new Circle(20);
```

In Python, we can only define an object on the heap

```
c = Circle(10)
```

Python allocates objects using object references which are essentially the same as passing by pointer

- Every object is allocated on the heap and it is then pointed to by an object reference All variables in pyton are object references

When we call a method, Python passes the object reference to the self parameter So the self parameter always refers to the item that's left of the dot

When you define a class, Python creates a special "class" object that holds the class's details. The class object only holds information about the overall Thing class, NOT individual Thing objects If you define a variable in the class and not the constructor of the class, it's value is associated with the overall class:

```
class Thing:
    thing_count = 0 # this defines a class member variable. Its value is associated with the overall class. It is also known as a class variable
    def __init(self):
        self.value = v
        Thing.thing_count _= 1
        self.thing_num = Thing.thing_count
    # when we use the class's name as a prefix to access a variable (Thing), this
refers to the overall class variable
```

Only use a class variable when you need a variable whose value is shared across all of your object instances

In Python Assignment of object references does NOT make a copy of the object like in C++ To make a distinct copy of an object, you need to use a special copying method

```
c2 = copy.deepcopy(c)
```

Python has deep copying: A deep copy makes a copy of the top-level object and every object referred to directly or indirectly by the top-level object

```
y = copy.deepcopy(x)
```

Python has shallow copying: A shallow copy just makes a copy of the top level object

```
y = copy.copy(x)
```

In Python, you don't have to worry about freeing objects when you're done with them, like you have to in C++! Why? Python automatically tracks each object and deletes it when your program no longer refers to it!

- As long as some variable refers to an object, the object is kept around
- But when the last reference to the object goes away, it becomes a candidate for garbage collection

Classes and Destructors

Python classes may have destructors - but they're rarely used, and NOT guaranteed to run! You can use the **del** destructor When an object is no longer referred to by an object reference, it may be garbage collected If so, during garbage collection, the destructor will be called automatically.

In garbage collected languages we use a destructor to free non-memory resources - like deleting temporary files, or closing network connections... But, since the destructor may never be called, it's best to define your own disposal function and explicitly call it when necessary.

Inheritance

How does a derived class inherit from a base class? By placing the base class in the parens (e.g. Student(Person)) How does a derived class method override a base class method? “Virtual by default” - as long as they have the same name! How does a derived method call a base-class method? Using the super() prefix!

```
class Person:
    def __init__(self, name):
        self.name = name
    def talk(self):
        print('Hi!\n')
```

```
class Student(Person):
    def __init__(self, name):
        super().__init__(name) # Don't forget to have the derived construcotr 
explicitly call the base class constructor
        self.units = 0
    def talk(self):
        print(f"Heya, I'm {self.name}.")
        print("Let's party! Oh... and")
        super().talk()

def chat(p):
    p.talk()
def cs131_lecture():
    s=Student('Angelina')
    chat(s)
```

Methods are virtual by default in Python, you may redefine any method in the derived class A derived class method can call any base class method via the super() prefix When you call a method, python automatically figures out which one to call

Duck Typing

If an object has the requested method, a call to it will just work. Doesn't have to be called directly from the object itself.

```
class PersonInDuckSuit:
  ...               # code omitted for clarity
  def quack(self):
    print('Hi! Err... oops, I mean quack quack.')

class Duck:
  ...               # code omitted for clarity
  def quack(self):
    print('Quack quack quack!')

class Vehicle:
  ...               # code omitted for clarity
  def drive(self):
    print('Vrooooom!')
def quack_please(x):
  x.quack()

p = PersonInDuckSuit()
d = Duck()
v = Vehicle()
quack_please(p)
quack_please(d)
quack_please(v)
```

This is a feature called duck typing - at runtime (and only when the line runs), Python checks if the input to quack_please() has a quack method. If it does, it runs that method; if not, an error is run.

Object identity in Python

== operator tests the equality of value is operator tests if the two object references refer to the same exact object in memory == looks at values, not references is looks at references, not values

Every distinct object has a unique ID number or "identity" in Python - you can ask for this with the id() function

Each object has a unique Id in python, so even if you are just adding number `Python booger = booger + 1` It will point to a new object in memory with that new incremented value every time

Python None Keyword: To see if an object reference refers to None, use the is (___) None syntax

Strings

In python, each string is an object just like our Circle objects! Strings are immutable, meaning they can't be modified once created

Lists

Lists are objects in Python and they support the same operations as strings, lists are mutable

- can hold multiple different types of values
- stuff.append('are') mutates the list
- stuff = stuff + ['are'] creates an entirely new list object due to the assignment operator

Lists are implemented accessing the jth element of a list x[j] is super fast

Tuples:

immutable, ordered groups of items, commonly used for returning multiple values from a function

- Can define a tuple explicitly with parentheses...or implicitly with commas The tup variable refers to the full returned tuple and then we can index the tuple's values

  ```
  return ('UCLA', 0)
  return 'USC', 100000
  ```

  ### Sets: an Abstract data type (ADT)

  Python sets hold a single copy of each unique value Sets are not ordered

Dictionaries

Python has first-class support for dictionaries (maps) Dictionaries are not lexicographically ordered: ordered based on the order they were inserted into the map You cannot have more than one key A dictionary holds a single copy of each unique value different keys map to different values

Parameter Passing

"Pass by object reference" same thing as pass by pointer in C++

Every variable in Python is an object reference - it just holds the address of a value The value could be a rectangle object, an integer, a string, a list, a tuple When we call a function, Python just passes the object reference (pointer) to the function! The parameter is also an object reference since it's just a variable Parameter always refers to the original value

Handling Errors in Python When Python encounters an error that it doesn't know how to handle it generates a special error called an "exception"

- If you don't add code to handle an exception, it will cause the program to terminate Can even have multiple except blocks dealing with a different type of issue

Multi-threading in Python

Each Python thread runs, it claims exclusive access to Python's memory/objects So only one thread generally does computation at a time Python has GIL (Global Interpreter Lock)

- can only have one owner at a time
- once a thread takes ownership of the GIL it's allowed to read/write Python objects
-
    - After a thread runs for a while, it releases the GIL to another thread and gives it ownership
- If a thread is waiting for the GIL, it simply falls asleep until it gets its turn
- Functional Influences: Comprehensions, Lambdas, Each lambda's body is a single expression on a single line and may refer to the lambda's parameters or variables from the current scope The lambda returns the value of the expression - no "return" statement is needed.